CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Transpiling Applications into Optimized Serverless Orchestrations

Short Paper

Joel Scheuner

Philipp Leitner

Joel Scheuner

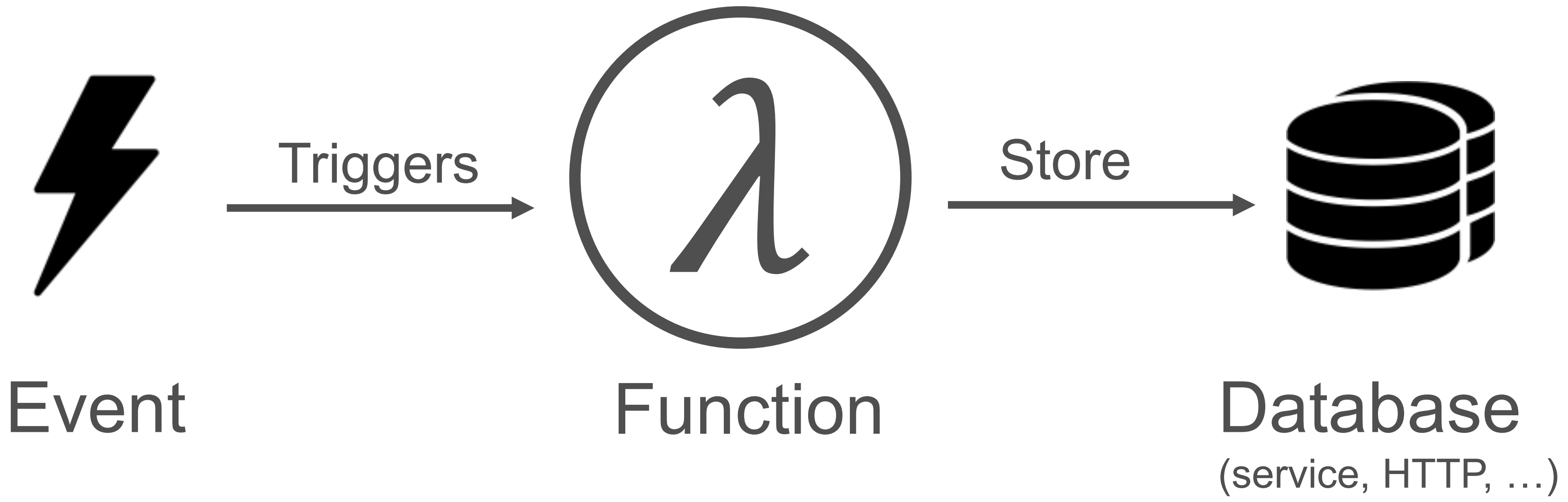
scheuner@chalmers.se

joe4dev

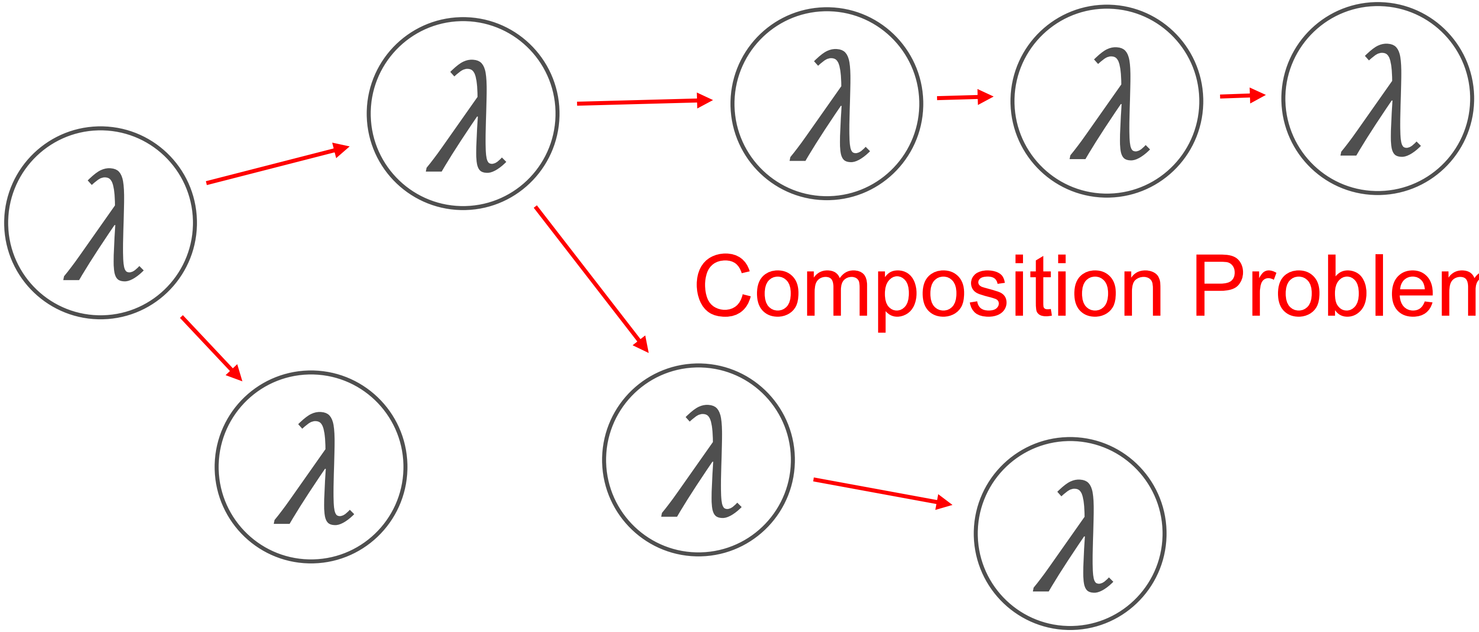
@joe4dev

Supported by WASP WALLENBERG AI, AUTONOMOUS SYSTEMS AND SOFTWARE PROGRAM

What is Serverless Computing?



Serverless Application



Composition Problem

Functions into apps

"I want to sequence functions"

"I want to run functions in parallel"

"I want to select functions based on data"

"I want to retry functions"

"I want try/catch/finally"

"I have code that runs for hours"

Source: AWS re:Invent 2016: NEW LAUNCH! Serverless Apps with AWS Step Functions (SVR201)

Composition Problem

"We need better orchestration for serverless workflows to make system design more straightforward and easier to implement"
Lessons learned experimenting with an AWS Lambda orchestration engine, blog 2017 by Ben Kehoe

"I'm looking for better ways to compose and re-use functions and serverless resources, cloudformation just doesn't cut it"
My wish list for AWS Lambda in 2018, blog 2018 by Yan Cui

"We don't yet have the Rails of serverless—something that doesn't necessarily expose that it's actually a Lambda function under the hood."
Serverless is eating the stack and people are freaking out—as they should be, blog 2018 by Forrest Brazeal

"composition and testing of functions […] sparsely covered by current scientific literature but […] immensely important in practice"
A mixed-method empirical study of Function-as-a-Service software development in industrial practice, JSS 2019

"serverless frameworks need to provide a way for tasks to coordinate"
Cloud Programming Simplified: A Berkeley View on Serverless Computing, technical report 2019

"Research will need to focus on what composition models would fit FaaS, on ways to express these compositions of functions, and on how to support (frequent) function-updates and hybrid-cloud deployment."
The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures, WOSC 2017

The Serverless Trilemma (ST)

ST-safe iff:

1. Functions considered as black boxes
2. Compositions of functions should be functions themselves
3. No double billing

The Serverless Trilemma – Function Composition for Serverless Computing, Onward! 2017

The Serverless Trilemma

Function Composition for Serverless Computing

Ioana Baldini
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
ioana@us.ibm.com

Perry Cheng
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
perry@us.ibm.com

Stephen J. Fink
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
sjfink@us.ibm.com

Nick Mitchell
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
nickm@us.ibm.com

Vinod Muthusamy
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
vmuthus@us.ibm.com

Rodric Rabbah
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
rabbah@us.ibm.com

Philippe Suter
Two Sigma Investments, LP
New York, NY, USA
philippe.suter@gmail.com

Olivier Tardieu
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
tardieu@us.ibm.com

Abstract

The field of serverless computing has recently emerged in support of highly scalable, event-driven applications. A serverless application is a set of stateless *functions*, along with the events that should *trigger* their activation. A serverless runtime allocates resources as events arrive, avoiding the need for costly pre-allocated or dedicated hardware.

While an attractive economic proposition, serverless computing currently lags behind the state of the art when it comes to *function composition*. This paper addresses the challenge of programming a composition of functions, where the composition is itself a serverless function.

We demonstrate that engineering function composition into a serverless application is possible, but requires a careful evaluation of trade-offs. To help in evaluating these trade-offs, we identify three competing constraints: functions should be considered as *black boxes*; function composition should obey a *substitution principle* with respect to synchronous invocation; and invocations should not be *double-billed*.

Furthermore, we argue that, if the serverless runtime is limited to a *reactive core*, *i.e.* one that deals only with dispatching functions in response to events, then these constraints form the *serverless trilemma*. Without specific runtime support, compositions-as-functions must violate at least one of the three constraints.

Finally, we demonstrate an extension to the reactive core of an open-source serverless runtime that enables the sequential composition of functions in a trilemma-satisfying way. We conjecture that this technique could be generalized to support other combinations of functions.

CCS Concepts • **Software and its engineering → Cloud computing**; *Organizing principles for web applications*;

Keywords cloud, serverless, functional, composition

ACM Reference Format:
Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3133850.3133855

1 Introduction

Under economic pressure to innovate ever more rapidly, organizations routinely exploit cloud computing rather than purchase hardware and operate data centers. Serverless computing, also known as *functions-as-a-service*, has recently emerged in support of highly scalable, event-driven applications in the cloud. It allows developers to write short-running, stateless *functions* that can be triggered by events generated from middleware, sensors, services, or users.

The serverless paradigm was pioneered by Amazon with the introduction of Lambda [Cross 2016], and today every major cloud provider offers a serverless platform [Apache 2016; Google 2016; Microsoft 2016]. The model appeals to many developers since it lets them focus on their application

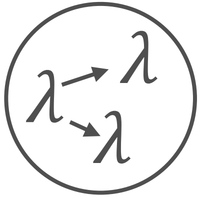
Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133855

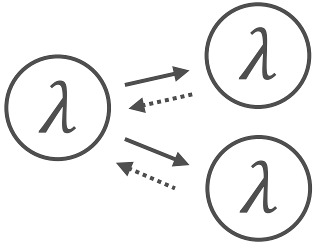
89

Composition Approaches

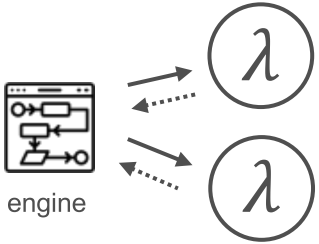
Function Fusion



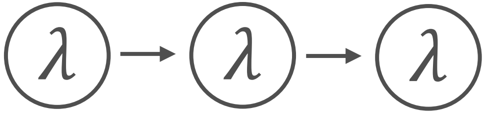
Function Coordinator



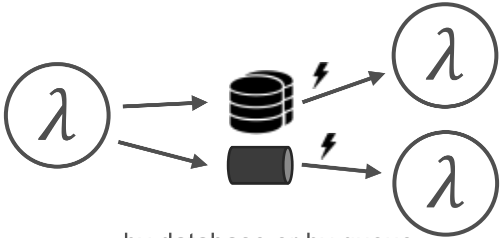
Function Workflows



Function Chaining



Event-Driven Function Composition



by database or by queue

Background:
Function Composition in a Serverless World, Kubeconf 2018
Serverless Apps with AWS Step Functions, AWS re:invent 2016

Function Orchestration Systems

AWS Step Functions

```
{
  "Comment" : "A demo Sequence state machine",
  "StartAt" : "f1",
  "States" : {
    "f1" : {
      "Next" : "f2",
      "Resource" :
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Type" : "Task"
    },
    "f2" : {
      "Next" : "f3",
      "Resource" : "[…]",
      "Type" : "Task"
    },
    "f3" : {
      "End" : true,
      "Resource" : "[…]"
      "Type" : "Task"
    }
  }
}
```

Azure Durable Functions

```
df.orchestrator(function*(context) {
    const parallelTasks = [];

    // Get a list of N work items to process in parallel.
    const workBatch = yield context.df.callActivity("F1");
    for (let i = 0; i < workBatch.length; i++) {
        parallelTasks.push(context.df.callActivity("F2", workBatch[i]));
    }

    yield context.df.Task.all(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    const sum = parallelTasks.reduce((prev, curr) => prev + curr, 0);
    yield context.df.callActivity("F3", sum);
});
```

Apache OpenWhisk Composer

```
composer.let({
        n: 224
    },
  composer.while(params =>
    n % 2 === 0,
      params => { n /= 2 }),
    composer.function(params =>
  console.log(`n=${n}`))
);
```

fission workflows

```
output: ExtractResult
tasks:
  Fib:
    run: repeat
    inputs:
      times: "{ param() || 0 }"
      do:
        run: javascript
        inputs:
          _prev:
            fn1: 0
            fn2: 1
          args:
            fn1: "{ task().Inputs._
            fn2: "{ task().Inputs._
          expr: "({
            'fn1': fn2,
            'fn2': (fn1 + fn2)
          })"
```

Comparison of Production Serverless Function Orchestration Systems, 4th WoSC 2018

Function Orchestration Systems

AWS Step Functions

```
{
  "Comment" : "A demo Sequence state machine",
  "StartAt" : "f1",
  "States" : {
    "f1" : {
      "Next" : "f2",
      "Resource" :
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Type" : "Task"
    },
    "f2" : {
      "Next" : "f3",
      "Resource" : "[…]",
      "Type" : "Task"
    },
    "f3" : {
      "End" : true,
      "Resource" : "[…]"
      "Type" : "Task"
    }
  }
}
```

Azure Durable Functions

```
df.orchestrator(function*(context) {
    const parallelTasks = [];

    // Get a list of N work items to process in parallel.
    const workBatch = yield context.df.callActivity("F1");
    for (let i = 0; i < workBatch.length; i++) {
        parallelTasks.push(context.df.callActivity("F2", workBatch[i]));
    }

    yield context.df.Task.all(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    const sum = parallelTasks.reduce((prev, curr) => prev + curr, 0);
    yield context.df.callActivity("F3", sum);
});
```

Apache OpenWhisk Composer

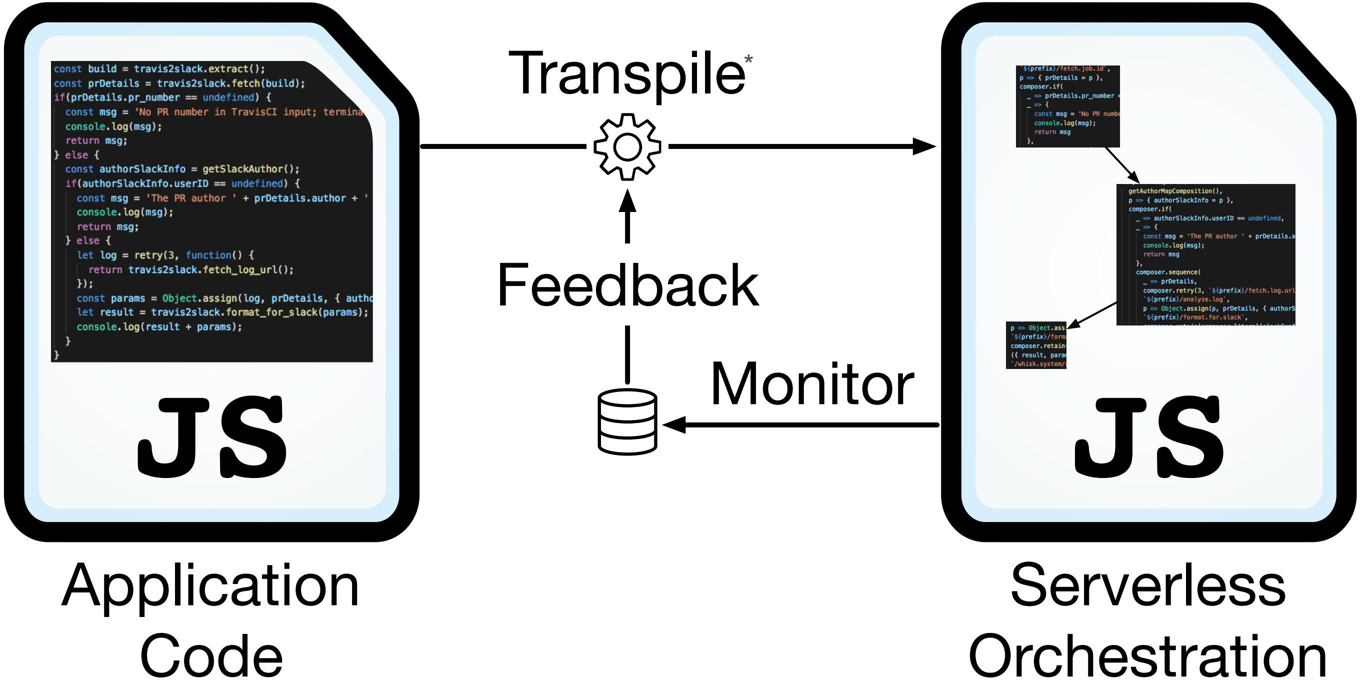
```
composer.let({
    n: 224
  },
  composer.while(params =>
    n % 2 === 0,
    params => { n /= 2 }),
  composer.function(params =>
)
```

fission workflows

```
output: ExtractResult
tasks:
  Fib:
    run: repeat
    inputs:
      times: "{ param() || 0 }"
      do:
        run: javascript
        inputs:
          _prev:
            fn1: 0
            fn2: 1
          args:
            fn1: "{ task().Inputs._
            fn2: "{ task().Inputs._
          expr: "({
            'fn1': fn2,
            'fn2': (fn1 + fn2)
          })"
```

Function-focus ➔ Application-focus

Composition Vision



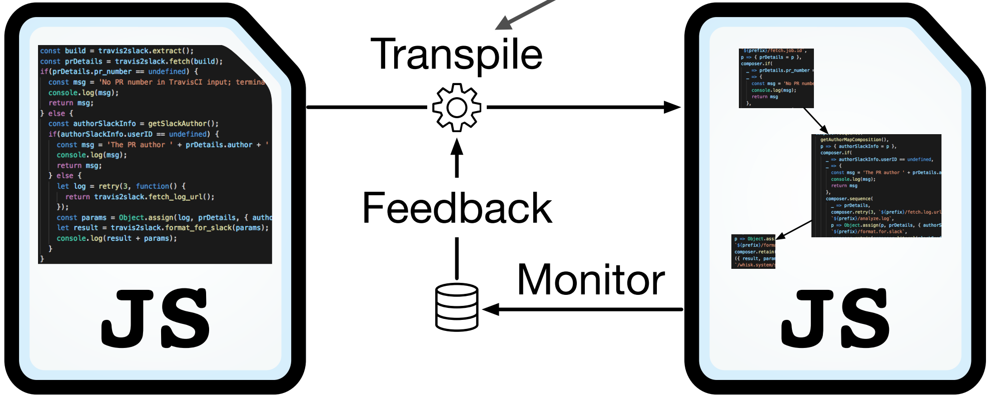
*source-to-source transformation of the abstract syntax tree (AST)

Prototype Implementation

Facebook jscodeshift with recast

facebook / jscodeshift
Code | Issues 57 | Pull requests 16
A JavaScript codemod toolkit.
338 commits | 1 branch

JS
Javascript



Application Code

Serverless Orchestration

Apache OpenWhisk Composer

apache / incubator-openwhisk
Watch 238 | Star 4,052 | Fork 783
Code | Issues 409 | Pull requests 26 | Projects 0 | Wiki | Security | Insights
Apache OpenWhisk is a serverless event-based programming service and an Apache Incubator project. https://openwhisk.apache.org/
serverless faas apache docker functions-as-a-service serverless-architectures serverless-framework openwhisk scala

APACHE OpenWhisk™

IBM Cloud Functions

apache / incubator-openwhisk-composer
Code | Issues 7 | Pull requests 0 | Sec
Composer is a new programming model for composi
109 commits | 1 branch

Transpilation Example

```
function f1() {
    return { message: 'f1' };
}
```

```
function f2() {
    return { message: 'f2' };
}
```

```
function f3() {
    return { message: 'f3' };
}
```

```
f1();
f2();
f3();
```

→

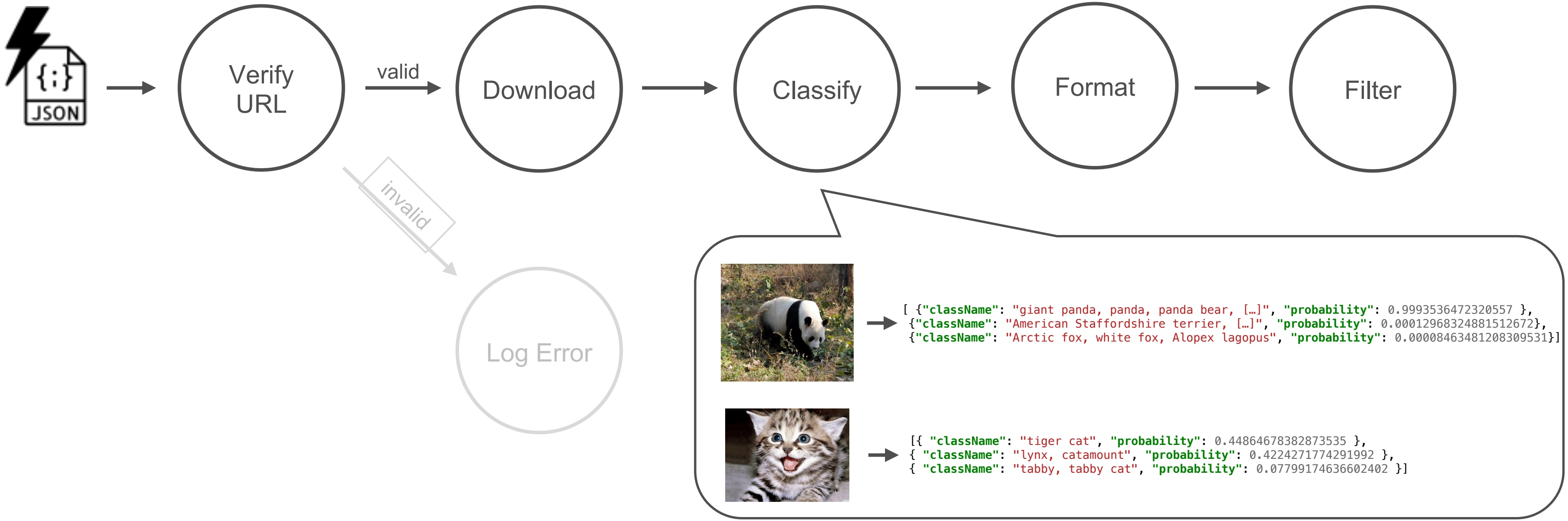
```
composer.sequence(
    composer.action('f1', { action: f1 }),
    composer.action('f2', { action: f2 }),
    composer.action('f3', { action: f3 })
);
```

```
var value = 6;
if (value % 2 === 0) {
    console.log(value / 2);
}
```

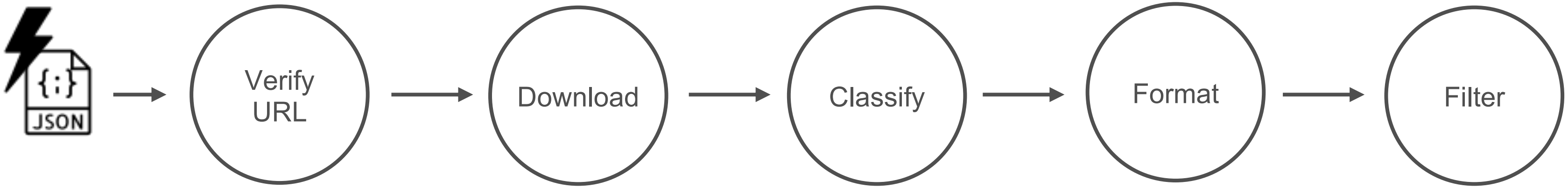
→

```
composer.let({
    value: 6
}, composer.if(params => value % 2 === 0,
params => console.log(value / 2)));
```

Composition Example Visual Recognition Application



Example Transformation

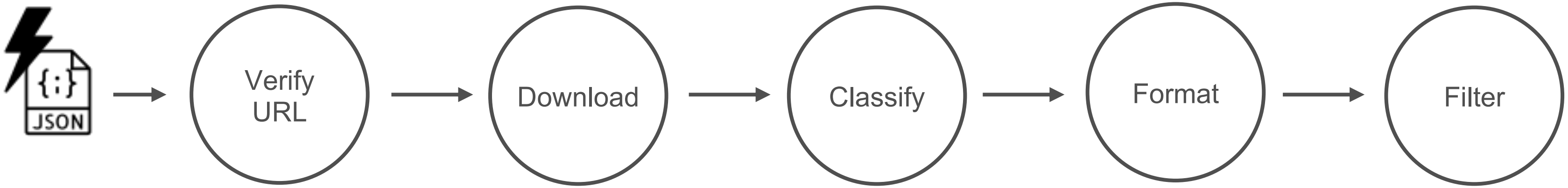



```
if(verifyUrl(url)) {
    var img = download(url);
    var prediction = classify(img);
    var label = format(prediction);
    return filter(result);
} else {
    return logError();
}
```

```
composer.if(composer.action('verifyUrl', { action: verifyUrl }),
  composer.sequence(
        composer.action('download', { action: download }),
        composer.action('classify', { action: {
           kind: 'blackbox’,
           image: 'jamesthomas/action-nodejs-v8:tfjs’,
           code:  `const main = ${classify}`,
           memory: 512 } }),
        composer.action('format', { action: format }),
        composer.action('filter', { action: filter })
    ), composer.action('logError', { action: logError })
  ),
```

Composition Performance (1)

IBM Cloud Functions



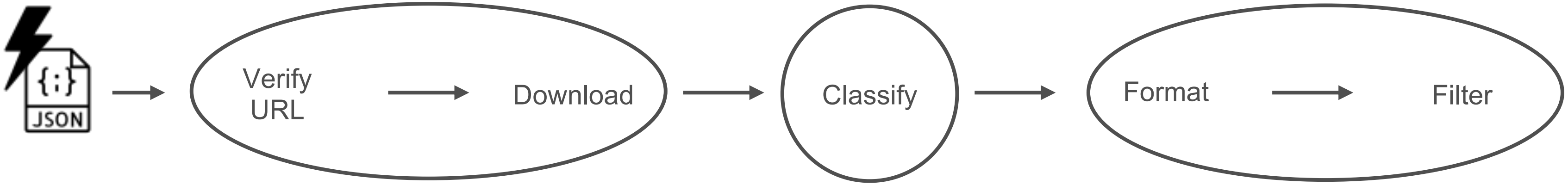
Execution Time* [ms]

Cold	300	1200	1300	300	300
Warm	2	600	700	2	2

*exemplary measurements

Composition Performance (2)

IBM Cloud Functions



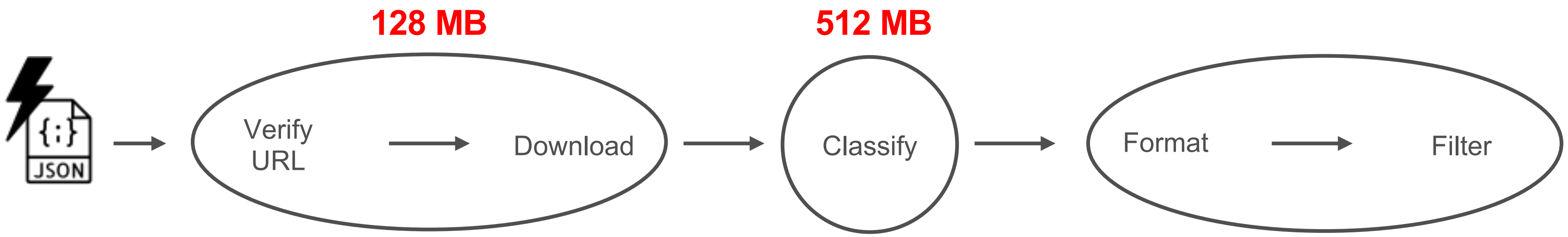
Execution Time* [ms]

Cold	1400	1300	(warm) 4
Warm	600	700	4

*exemplary measurements

Composition Cost

IBM Cloud Functions



Monthly costs* [USD]

128	1.3
256	2.6
512	5.2

4x

*based on 1’000’000 warm-start requests per month

**Pricing based on: https://cloud.ibm.com/openwhisk/learn/pricing
$0.000017 per second of execution, per GB of memory allocated

Benefits

- More accessible to build serverless applications
  - Transpilation from generic JS to platform-specific code
- Faster application runtime
  - Automated function fusion
- Cheaper computation cost
  - Targeted function size

Limitations

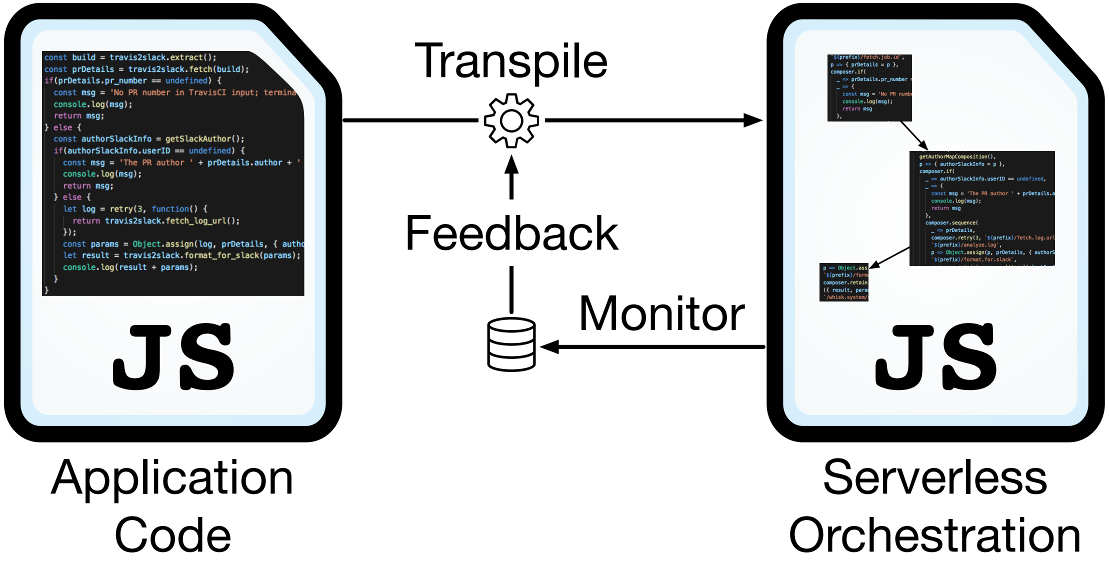
- Function fusion only when code available → violating ST black-box constraint
- Harder to debug at runtime
- Data marshalling overhead and limitations
- Integration into third party services

Future Work

- Extend transpilation prototype
  - Support more composition primitives
- Integrate and evaluate dynamic deployment alternatives

Conclusion

scheuner@chalmers.se



Thursday 9:00 – 10:30 in N440:

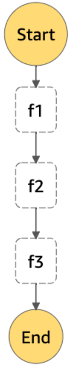
Tutorial 5: Performance Benchmarking of Infrastructure-as-a-Service (IaaS) Clouds with Cloud WorkBench

Discussion Input

How should serverless compositions be expressed?

As data?

```
{
  "Comment" : "A demo Sequence state machine",
  "StartAt" : "f1",
  "States" : {
    "f1" : {
      "Next" : "f2",
      "Resource" : "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Type" : "Task"
    },
    "f2" : {
      "Next" : "f3",
      "Resource" : "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Type" : "Task"
    },
    "f3" : {
      "End" : true,
      "Resource" : "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Type" : "Task"
    }
  }
}
```




```
final StateMachine stateMachine = stateMachine()
        .comment("A demo Sequence state machine")
        .startAt("f1")
        .state("f1", taskState()
                .resource("arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME")
                .transition(next("f2")))
        .state("f2", taskState()
                .resource("arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME")
                .transition(next("f3")))
        .state("f3", taskState()
                .resource("arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME")
                .transition(end()))
        .build();
```

As code?

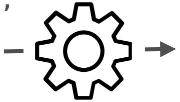
```
module.exports = composer.sequence(
    composer.action('f1'),
    composer.action('f2'),
    composer.action('f3'),
);
```

```
f1();
f2();
f3();
```

Should machines decide upon deployment structure?

- Is is practical (e.g., understandable) to have dynamically changing deployment structures?
  - Debugging (source maps)?
  - Testing?

```
if(verifyUrl(url)) {
    var img = download(url);
    var prediction = classify(img);
    var label = format(prediction);
    result = filter(result);
} else {
    logError();
}
```

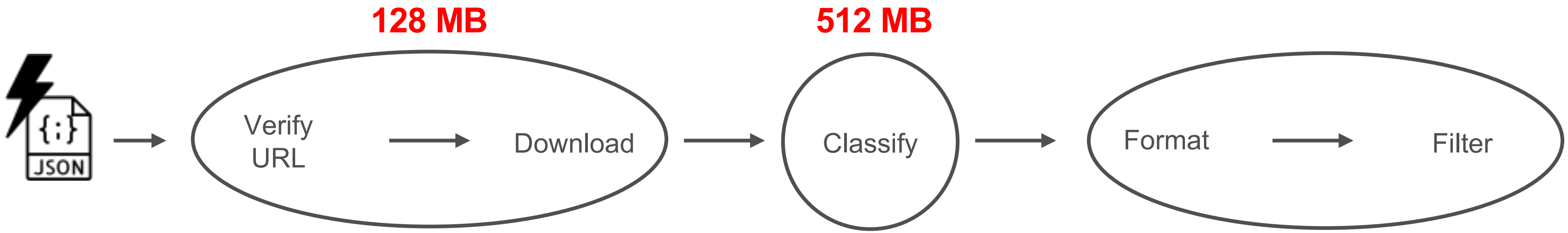


```
composer.if(composer.action('verifyUrl', { action: verifyUrl }),
  composer.sequence(
        composer.action('download', { action: download }),
        composer.action('classify', { action: { kind: 'blackbox', image:
'jamesthomas/action-nodejs-v8:tfjs', code:  `const main = ${classify}`, memory: 512 }
}),
        composer.action('format', { action: format }),
        composer.action('filter', { action: filter })
    ), composer.action('logError', { action: logError })
  ),
```

Entering composition[1].consequent[2]"

Which application types benefit from this approach?

- Which applications have heterogenous-enough footprints?



Any related work from (other) communities?

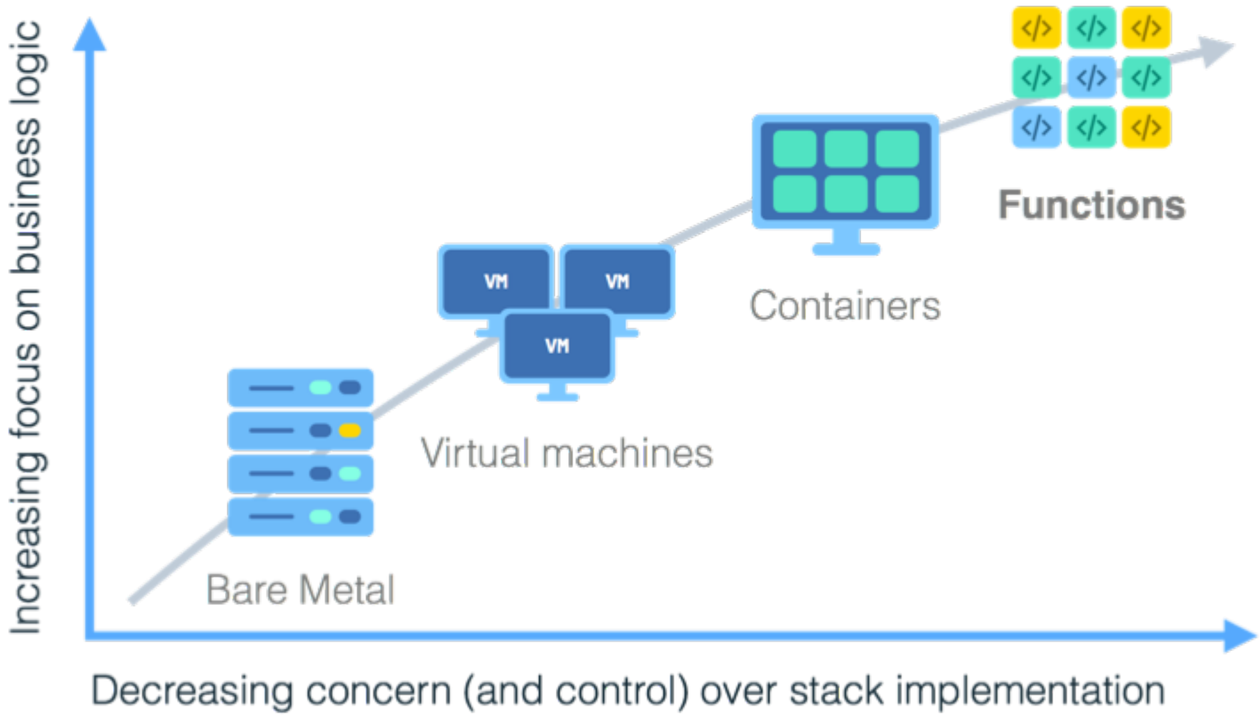
- Programming Languages (PL)
- Domain Specific Languages (DSL)
- Workflows
- …

scheuner@chalmers.se

UNIVERSITY OF
GOTHENBURG

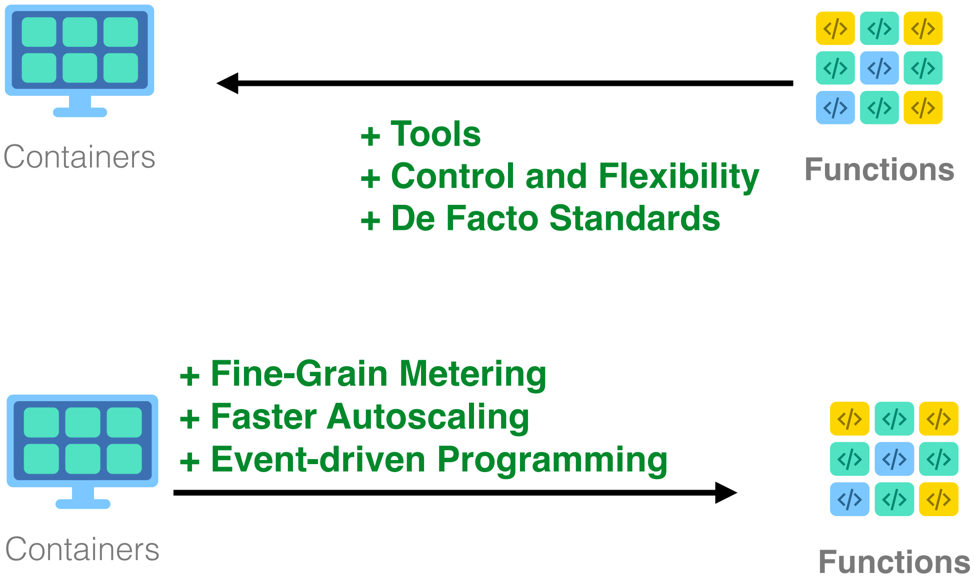
CHALMERS
UNIVERSITY OF TECHNOLOGY

Serverless Background



Source: © 2018 IBM Corporation

Serverless Pros and Cons



Source: © 2018 IBM Corporation

Serverless Application Types

Serverless is good for

short-running
stateless
event-driven

- Microservices
- Mobile Backends
- Bots, ML Inferencing
- IoT
- Modest Stream Processing
- Service integration

Serverless is not good for

long-running
stateful
number crunching

- Databases
- Deep Learning Training
- Heavy-Duty Stream Analytics
- Numerical Simulation
- Video Streaming

Source: Slides Workshop of Serverless Computing (WoSC'4), 2018

Abstract Syntax Tree (AST)

```
var value = 6;
if (value % 2 === 0) {
    console.log(value / 2);
}
```

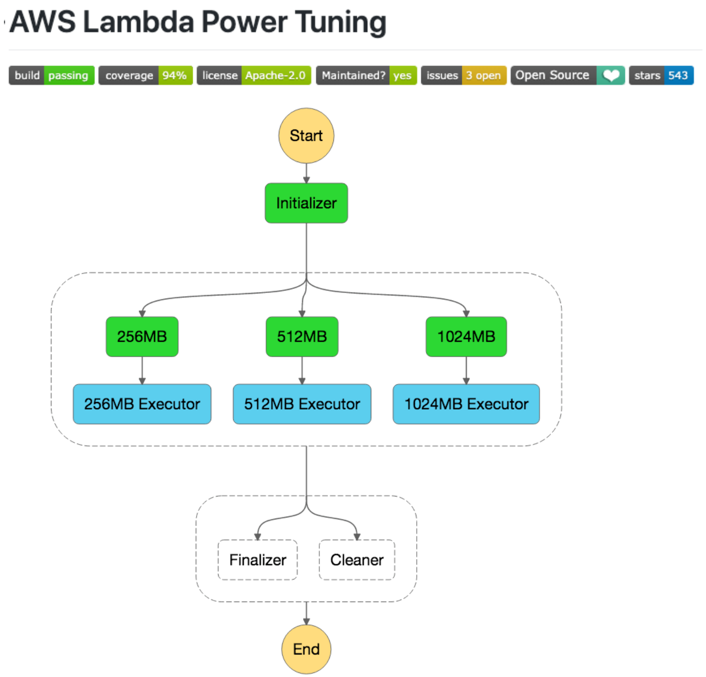
```
+ VariableDeclaration {declarations, kind}
- IfStatement {
  - test: BinaryExpression {
      operator: "==="
    - left: BinaryExpression {
        operator: "%"
      - left: Identifier = $node {
          name: "value"
        }
      + right: Literal {value, raw}
      }
    + right: Literal {value, raw}
    }
  + consequent: BlockStatement {body}
  }
```

Tree Visualization using AST Explorer: https://astexplorer.net/

AST Transformation Example

```
function transform(file, api, options) {
    imports.register(j, imports.config.CJSBasicRequire);
    const { statement } = j.template;
    const parsed = j(file.source)
    parsed.find(j.CallExpression)
        .replaceWith(function (path) {
            const actionName = path.value.callee.name;
            const left = j.memberExpression(
                j.identifier('module'),
                j.identifier('exports')
            )
            const right = j.callExpression(
                j.memberExpression(
                    j.identifier('composer'),
                    j.identifier('action')
                ),
                [
                    j.literal(actionName),
                    createActionReference(actionName)
                ]
            )
            return j.assignmentExpression(
                        '=',
                        left,
                        right,
                    )
        })
    const transformed = parsed.addImport(statement`
        const composer = require('openwhisk-composer');
    `)
    const outputOptions = {
        quote: 'single'
    }
    return transformed.toSource(outputOptions);
}
```

AWS Lambda Power Tuning



https://github.com/alexcasalboni/aws-lambda-power-tuning